6.0001 Problem Set 1

Handed out: Wed, September 8, 2021
Due: Monday, September 20, 2021 at 9PM
Checkoff due: Tuesday, September 28, 2021 at 9PM

Objectives
- Introduction to control flow in Python
- Formulating a computational solution to a problem
- Exploring bisection search

Overview
- Complete each of the **three** problems in the respective python files: **ps1a.py**, **ps1b.py** and **ps1c.py**
- Include **comments** to help us understand your code—read the **[Style Guide](#)** posted on Stellar for more specific instructions regarding this.
- **ps1_tester.py** is a tester file that you may run to test your code. **Make sure ps1_tester.py is in the same folder as ps1a.py, ps1b.py, ps1c.py, and put_in_function.py.**

Collaboration
- Students may work together, but each student should write up and hand in their assignment separately. Students may not submit the exact same code, and we **do** check for plagiarism.
- Students are not permitted to look at or copy each other's code or code structure.
- Include the names of your collaborators in a comment at the start of each file.
- **Please refer to the collaboration policy in the [Course Information](#) for more details.**

Important Notes
- Read sections 1, 2, 3, and 4 of the [Style Guide](#).
- If you get an error like `ModuleNotFoundError: No module named 'ps1b_in_function"` at any point, restart the Spyder kernel.
- **Get user input and declare the appropriate variables under the comment headers in ps1a.py, ps1b.py, and ps1c.py exactly as specified.** Failure to do so (e.g. using different variable names than the ones that are specified or declaring them under the wrong comment headers) will **result in a score of 0** from the autograder.
- **Do not remove or change any of the comments that are provided in the original ps1a.py, ps1b.py, and ps1c.py files.**

Part A: Saving up for a house

You have just graduated from MIT and have found a job! After moving to the Bay Area, you realize that in order to someday afford your dream home, you have to start saving up for the down payment now.

> **Your goal in this section is to determine the number of months that it will take for you to save up for your down payment**. Your down payment is calculated by multiplying the *total cost* of your dream house by the *down payment percentage*.

User Inputs
Ask the user to enter the following variables and cast them as *floats*. They must be initialized in the following order at the very beginning of your program (before declaring other variables).
1. The starting annual salary (**annual_salary**)
2. The portion of the salary that will be saved (**portion_saved**). This variable should be in decimal form (e.g. 0.1 for 10%).
3. The cost of your dream home (**total_cost**)

Writing the Program
You will need to determine how many months it will take to save up for your down payment given the following information:
1. **annual_salary**, as described above.
2. **portion_saved**, as described above.
3. **total_cost**, as described above.
4. **portion_down_payment,** which is the percentage of the total cost needed for a down payment is. Assume that portion_down_payment = 0.20 (20%).
5. **current_savings,** which is the amount that you have saved thus far. This amount starts at $0.
6. **r**, which is your annual rate of return. In other words, at the **end of each month**, you will receive an additional `current_savings*r/12` dollars for your savings (the 12 is because r is an annual rate). Assume r = 0.04 (4%).
7. At the end of each month, your savings will increase by (1) a percentage of your monthly salary and (2) the monthly return on your investment. **(Note: the investment amount used to calculate the monthly return is the amount you had saved at the start of each month.)**

Your program should store the number of months required to save up for your down payment using a variable called **months**.

Notes

- Be careful about values that represent **annual** amounts versus **monthly** amounts.
- If the number of months your program returns is off by one, reread the **red** text above.
- Assume that users enter valid inputs (e.g. no string inputs when expecting a float).
- Your program should print outputs in the same format as the test cases below.

Test Case 1
Enter your annual salary: 112000
Enter the portion of your salary to save, as a decimal: .17
Enter the cost of your dream home: 750000
Number of months: 83

Test Case 2
Enter your annual salary: 65000
Enter the portion of your salary to save, as a decimal: .20
Enter the cost of your dream home: 400000
Number of months: 67

Test Case 3
Enter your annual salary: 350000
Enter the portion of your salary to save, as a decimal: .3
Enter the cost of your dream home: 10000000
Number of months: 171

Testing
Run **ps1_tester.py** in the same folder as **ps1a.py** and **put_in_function.py**. You should pass the first 3 test cases.

Part B: Saving with a raise

In Part A, we assumed that your salary did not change over time. However, you are an MIT graduate, and clearly you are going to be worth more to your company over time! In this part, we will build on your solution to Part A by **adding a salary raise every six months**. Copy your code from Part A into the corresponding sections in **ps1b.py**.

User Inputs

There is one additional user input in Part B. Remember to cast these inputs as *floats* and initialize them in the following order before declaring other variables.

1. The starting annual salary (**annual_salary**)
2. The portion of salary to be saved (**portion_saved**)
3. The cost of your dream home (**total_cost**)
4. The semi-annual salary raise (**semi_annual_raise**), which is a decimal percentage (e.g. 0.1 for 10%)

Writing the Program

Write a program to calculate how many months it will take for you to save up for a down payment. You can reuse much of the code from Part A. Like before, assume that your investments earn an annual rate of return **r** = 0.04 (or 4%) and that **portion_down_payment** = 0.20 (or 20%). In this version, **annual_salary** increases by **semi_annual_raise** at the **end of every six months**. Your program should store the number of months required to save up for your down payment using a variable called **months**.

Notes

- Be careful about values that represent **annual** amounts versus **monthly** amounts.
- Raises should only happen **at the end of** the 6th, 12th, 18th month, and so on.
- If the number of months your program returns is off by one, reread the **red** text above.
- Assume that users enter valid inputs (e.g. no string inputs when expecting a float).
- Your program should print outputs in the same format as the test cases below.

Test Case 1

Enter your starting annual salary: 110000
Enter the percent of your salary to save, as a decimal: .15
Enter the cost of your dream home: 750000
Enter the semi-annual raise, as a decimal: .03
Number of months: 80

Test Case 2

Enter your starting annual salary: 350000
Enter the percent of your salary to save, as a decimal: .3
Enter the cost of your dream home: 10000000

Enter the semi-annual raise, as a decimal: .05
Number of months: 118

Testing

Run **ps1_tester.py** in the same folder as **ps1b.py** and **put_in_function.py**. You should now pass the first 5 test cases.

Part C: Choosing an interest rate

In Part A and B, you explored how (1) the percentage of your salary saved each month and (2) a semi-annual raise affects how long it takes to save for a down payment given **a fixed rate of return**, **r**.

In Part C, we will have a **fixed initial deposit** and the ability to choose a value for the rate of return, r. Given an initial deposit amount, our goal is to find the **lowest** rate of return that enables us to save enough money for the down payment in 3 years.

User Inputs
Cast the user input as a *float* in the beginning of your program.
   1. The initial amount in your savings account (**initial_deposit**)

Writing the Program
Write a program to calculate what **r** should be in order to reach your goal of a sufficient down payment in 3 years, given an **initial_deposit**. To simplify things, assume:
   1. The cost of the house that you are saving for is $800,000
   2. The down payment is 20% of the cost of the house (i.e. $160,000)

Use the following formula for compounded interest in order to calculate the predicted savings amount given a rate of return **r**, an **initial_deposit**, and **months**:

```
current_savings = initial_deposit * (1 + r/12) ** months
```

You will use [bisection search](#) to determine the *lowest* rate of return, **r**, that is needed to achieve a down payment on a $800K house in 36 months. Since hitting this exact amount is a bit of a challenge, we only require that your savings be within $100 of the required down payment. For example, if the down payment is $1000, the total amount saved should be between $900 and $1100 (exclusive).

Your bisection search should update the value of **r** until it represents the *lowest* rate of return that allows you to save enough for the down payment in 3 years. **r** should be a float (e.g. 0.0704 for 7.04%). Assume that **r** lies somewhere between 0% and 100% (inclusive).

The variable **steps** should reflect the number of steps your bisection search took to get the best **r** value (i.e. **steps** should equal the number of times that you bisect the testing interval).

Notes
   ● There may be multiple rates of return that yield a savings amount that is within $100 of the required down payment on a $800,000 house. The grader will accept any of these values of r.

- **If the initial deposit amount is greater than or equal to the required down payment, then the best savings rate is** `0.0`.
- **If it is not possible to save within $100 of the required down payment in 3 years given the initial deposit and a rate of return between 0% and 100%, r should be assigned the value** `None`.
  - **Note**: the value `None` is different than `"None"`. The former is Python's version of a null value, and the latter is a string. `r` should be set to the former definition of `None`
- Depending on your stopping condition and how you compute the amount saved for your bisection search, your number of steps may vary slightly from the example test cases. Running ps1_tester.py should give you a good indication of whether or not your number of steps is close enough to the expected solution.
- If a test is taking a long time, you might have an infinite loop! Check your stopping condition.
- Your program should print outputs in the same format as the test cases below.

Test Case 1
Enter the initial deposit: 65000
Best savings rate: 0.30419921875 [or very close to this number]
Steps in bisection search: 10 [or very close to this number]

Test Case 2
Enter the initial deposit: 150000
Best savings rate: 0.021484375 [or very close to this number]
Steps in bisection search: 8 [May vary based on how you implemented your bisection search]

Test Case 3
Enter the initial deposit: 1000
Best savings rate: None
Steps in bisection search: 0 [May vary based on how you implemented your bisection search]

Testing
Run **ps1_tester.py** in the same folder as **ps1c.py** and **put_in_function.py** and you should pass all 8 test cases.

Hand-in Procedure

1. Time and Collaboration Info

At the start of each file, in a comment, write down the number of hours (roughly) you spent on the problems in that part, and the names of your collaborators. For example:

```
# Problem Set 1A
# Name: Jane Lee
# Collaborators: John Doe
# Time Spent: 3:30
# Late Days Used: 1 (only if you are using any)

… your code goes here …
```

2. Submit

Before you upload your files, remove all of your extra debugging print statements. In addition, open a new console in Spyder and rerun your programs. Be sure to run ps1_tester.py and make sure all the tests pass. The tester file contains a *subset* of the tests that will be run to determine the problem set grade.

To submit a file, upload it to the **Problem Sets link** **on the website**. You may upload new versions of each file until the **5PM** deadline, but anything uploaded after that time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission.

To submit a pset with multiple files, you may submit each file individually through the submission page. Be sure to title each submission with the corresponding problem number.

After you submit, please make sure the files you have submitted show up on the problem set submission page. When you upload a new file with the same name, your old one will be overwritten.

And that's it! Congrats on finishing your first 6.0001 pset :)